Final project: LabBook

Lucas Nesi

November 30, 2018

Contents

1 Introduction

The QR factorization of sparse matrices using the software package `QR_mumps` may present different behaviors regarding the selection of execution parameters for the StarPU runtime and the own `QR_mumps`. An erroneous selection of parameters values can negatively impact in the application performance, while a correct configuration could provide positive impacts using the same workload and hardware. The problem of parameter tuning is present in many applications usually because they represent options in heuristic parts of the application, like the StarPU scheduler, or in arbitrary constants, like the block size used for the matrix decomposition. The objectives of this report are the following. (i) For a set of selected parameters, indicate which ones had the most impact on the performance. (ii) Considering the selected options for each parameter, find the best and worst combination.

In this report, we select a set of parameters for the StarPU runtime and the `QR_mumps` application to conduct an experimental performance analysis over the different executions configurations. In a first step, we perform a preliminary test with all parameters using only two levels per factor to calculate the allocation of variance. Utilizing the factors with the most impact, we conduct the second step of experiments with all selected parameters levels in a full factorial style.

2 Methodology

The principal goal of this work is to investigate the performance impact of the executions parameters on the software package `QR_mumps` factorization step. For achieving this objective, we select a set of four parameters, two from the StarPU runtime, and two from the `QR_mumps` to conduct a series of experiments. The chosen parameters for this study as factors are the following.

- (i) StarPUs Scheduling, with levels `LWS` and `DMDA`, where the `LWS` is a simpler scheduler while the `DMDA` is a more sophisticated one; presenting very different options to evaluate this factor.
- (ii) StarPUs number of workers per GPU, from 0 to 4, the possible values for this parameter.
- (iii) `QR_mumps` block size, with levels 320, 640 and 960, common values used in many algebraic kernels.
- (iv) `QR_mumps` inner block size, with levels 32 and 64, this parameter requires to be a divider of the block size.

The selection of these specific four parameters has the objective to study the software in a well-defined environment. Our first hypotheses is that these parameters have a considerable impact on the performance, and this study will validate it. However, other execution parameters may impact on the application performance.

The input matrices used in this study are present in Table 1, and are essentially canonical sparse matrices available at SuiteSparse Matrix Collection[1]. All four selected input matrices come from combinatorial problems. The TF16 is the base workload for the experiments. We choose the TF17 because it is just a larger version of the TF16 matrix. The selection of CH7 is because it has a similar number of non-zeros was the TF16 matrix but its a bigger matrix, 35280 rows against 15437 and 52920 columns against 19321. The last application input, MK12 was also selected because of a similar number of non-zeros as TF16, but it has more rows, 51975, and fewer columns, 13860. The general reason for the selection of those matrices is that they have different characteristics to exploit the diverse values that the parameters have, and check if they behave equally in all situations.

Table 1: Selected matrices for workload

Matrix	Rows	Columns	Non-Zeros
TF16	15437	19321	216173
TF17	38132	48630	586218
CH7-7-b5	35280	52920	211680
MK12-b3	51975	13860	207900

The machines used in this work are part of the UFRGS INF GPPD group and are present in Table 2. The main difference between machines used in our study is the number of CPUs and the GPU versions. While `Draco 7` have 2 CPUs, `TUPI` have newer GPUs.

Table 2: Machines used in the experiments

Machine:	Tupi	Draco 7
CPU	Intel Xeon E5-2620	2 x Intel Xeon E5-2630
Memory	64 GB DDR4 RAM	128 GB DDR3 RAM
GPU	2 x NVIDIA GTX 1080Ti	2 x NVIDIA Tesla K20m
SO	Ubuntu 18.04.1	Ubuntu 18.04.1
Kernel	4.15.0-38-generic	4.15.0-38-generic
CUDA	10	10
Driver GPU	410.48	410.48
GCC	7.3.0	7.3.0

[1]SuiteSparse Matrix Collection Website: `https://sparse.tamu.edu/`

3 Experiments

The execution of the experiments require the source code of `QR_Mumps` that is not publicly available. Because of this, the following scripts are not reproducible without the software. Also, because I was in doubt about legal/license reasons, the data included in this repository is only data generated by the last script (Generate Tidy Data from experiments results) and the random experiments sequence generated.

3.1 Execute `QR_Mumps` script

The following script execute one `QR_mumps` experiment. The parameters for the execution are present on the first lines.

```bash
#!/bin/bash
# Execute one QR_MUMPS Experiment
# Execution:
# $1 - QR_MUMPS PATH - Like /scratch/llnesi/qrmumps
# $2 - SAVE PATH
# STARPU:
# $3 - Scheduler
# $4 - num GPUs
# $5 - Workers per GPU
# QR_MUMPS:
# $6 - Matrix
# $7 - Size of blocks
# $8 - IB (Other block size)
# $9 - Memory Limit


INSTALL_PATH=$1
SAVE_PATH=$2

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:\
        $INSTALL_PATH/lib:$INSTALL_PATH/lib64
export PATH=$PATH:$INSTALL_PATH/bin

export STARPU_SCHED=$3
export STARPU_FXT_TRACE=0
export STARPU_NCPU=5
export STARPU_NCUDA=$4
export STARPU_NWORKER_PER_CUDA=$5
#export STARPU_LIMIT_CPU_MEM=60000

export CONF_FILE=$INSTALL_PATH/conf.txt

echo "matfile         '$6'
qrm_ounit      6
qrm_eunit      6
qrm_dunit      0
qrm_ordering   4
qrm_mb         $7
qrm_nb         $7
qrm_ib         $8
qrm_bh         0
qrm_keeph      0
nrhs           1
qrm_rhsnb      1
qrm_mem_relax  $9
end" > $CONF_FILE

cp -r -u /home/users/llnesi/qr_mumps/mat /scratch/llnesi/
cp $CONF_FILE $SAVE_PATH
bash /home/users/llnesi/slurm_scripts/node_info.sh > $SAVE_PATH/node.txt
$INSTALATION_PATH/bin/dqrm_test < $CONF_FILE > $SAVE_PATH/result.txt
```

3.2 Run a batch of experiments on GPPD-Cluster

This is a sample sbatch script used to install and run the experiments.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --partition=tupi
#SBATCH --time=72:00:00
#SBATCH --output= ~/%x_%j.out
#SBATCH --error= ~/%x_%j.err
#SBATCH --chdir /scratch/$USER/

INSTALATION_PATH=/scratch/$USER/qr_mumps
BASE_SAVE=/home/users/$USER/tupi_qrmumps

srun -l -J "install" bash ~/install-spack.sh\
      "qr_mumps+starpu+cuda~fxt+metis ^starpu@develop" "$INSTALATION_PATH"

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:\
       $INSTALATION_PATH/lib:$INSTALATION_PATH/lib64
export PATH=$PATH:$INSTALATION_PATH/bin

# List of experiments
exper=()

for mat in "mk12-bt" "TF16" "TF17" "ch7-7-b5"; do
 for scheduler in lws dmda; do
   for gpu_worker in 0 1 2 3 4; do
      for size in 160 320 960; do
         for ib in 32 80 160; do
           for it in $(seq 1 30); do
             save=$BASE_SAVE"/${mat}_${scheduler}_${gpu_worker}_${size}_${ib}/${
  it}"
             gpus=2
             mkdir -p $save
             if [ "$gpu_worker" = "0" ]; then
                gpus=0;
             fi
             exper+=("$INSTALATION_PATH $save $scheduler"\
                     "$gpus $gpu_worker mat/${mat}.mtx $size $ib -1")
           done
         done
      done
   done
 done
done

readarray final_exp < <(shuf -e "${exper[@]}")
echo ${final_exp[@]} > $BASE_SAVE/exp.txt

for ex in "${final_exp[@]}"; do
   echo "Running $ex"
   srun bash ~/execute_qrmumps_tupi.sh $ex
done
```

3.3 Generate Tidy Data from experiments results

This script access the save folder generate by the earlier script and collect the factorization time reported by the `QR_Mumps`.

```
#!/bin/bash
#$1 - Experiment result folder
cd $1
echo "scheduler;workersgpus;blocksize;innerblock;time"
for file in $(find | grep result.txt); do
   t=$(cat $file | grep "Time to do the facto" |\
     sed -e 's/[ ]*Time to do the facto[ ]*:[ ]*\(.*\)/\1/');
```

```
    v=$(echo $file | sed -e 's/\.\/\(.*\)\/[0-9]*\/result.txt/\1/'|\
        sed -e 's/_/;/g');
    echo $v";"$t;
done
```

Example of the script execution:

```
bash extract_qrmumps_times.sh results_folder
```

Example of output:

```
scheduler;workersgpus;blocksize;innerblock;time
lws;1;960;64;6.692E+01
lws;1;960;64;6.703E+01
lws;1;960;64;6.730E+01
lws;1;960;64;6.814E+01
lws;1;960;64;6.836E+01
lws;1;960;64;6.658E+01
lws;1;960;64;6.701E+01
lws;1;960;64;6.647E+01
lws;1;960;64;6.695E+01
lws;1;960;64;6.730E+01
dmda;4;960;64;6.059E+01
dmda;4;960;64;6.040E+01
dmda;4;960;64;6.028E+01
dmda;4;960;64;6.035E+01
dmda;4;960;64;6.028E+01
dmda;4;960;64;6.043E+01
dmda;4;960;64;6.025E+01
dmda;4;960;64;6.035E+01
dmda;4;960;64;6.051E+01
dmda;4;960;64;6.018E+01
```

We save it as a csv to be loaded in R.

4 Data Analysis

As previously mentioned, we divided our experiments into two steps; (i) Conducting a $2^k n$ experiment setup to find the most impacting parameters; (ii) For a subset of parameters, selected based on the first step of results, conduct a full factorial experiment.

4.1 First Step

The first step was executed with the following parameters:

- Scheduler: `DMDA`, `LWS`.
- Number of GPU workers: 0, 1.
- Block Size: 320, 640.
- Inner Block Size: 32, 64.

After the execution of the previous mentioned scripts, the files `tf17_draco7.csv`, `tf17_tupi.csv` and `tf16_draco7.csv` were generated. They contain the execution with the TF16 matrices on `draco7` and TF17 on `draco7` and `TUPI`. Using the guidelines of Raj Jain, in his book The art of computer systems performance analysis, chapter 18, we construct the allocation of variation table.

4.1.1 Load Libraries

This will load the Tidyverse library. The code `options(crayon.enabled FALSE)` disable the tidyverse color output, a problem found using emacs.

```
options(crayon.enabled = FALSE)
library(tidyverse)
```

4.1.2 Load Data

Each CSV file contains all the measurements for all experimental factors for one matrix on one machine. Each line corresponds to one execution, and have following five elements in order, the scheduler used, the number of GPUs workers, the block size, the inner block size, and the measurement time for the factorization step.

The first step is to create a reader function that will be used to read the generated files from the script and load all three files on respective variables. The types for each column are:

- Scheduler : Character
- WGPUS: (Workers per GPU) : Integer
- Size (Block Size) : Integer
- IB (Inner Block Size) : Integer
- Time (Factorization Time) : Double

```
read_measurements <- function(file){
    read_delim(file, delim=";",
        col_names=c("Scheduler", "WGPUS", "Size", "IB", "Time"),
        col_types=c(Scheduler = col_character(),
                    WGPUS = col_integer(),
                    Size = col_integer(),
                    IB = col_integer(),
                    Time = col_double()
        ))
}

data_tf17_draco <- read_measurements("data/tf17_draco7.csv");
data_tf17_tupi <- read_measurements("data/tf17_tupi.csv");
data_tf16_draco <- read_measurements("data/tf16_draco7.csv");
```

4.1.3 Create Allocation of Variation Table for 4 factors Function

Now, following the specifications of the book The art of computer system performance analysis, we construct a function to generate the allocation of variation table from an input data set. The function is projected to compute with $k=4$. The function selected values are responsible for selecting only the two desired levels per factor when passing the values -1 or 1 from the allocation of variation table. In this case, when factor A (the scheduler) values 1, we search for entries with `DMDA` scheduler; and when it is -1, we search for entries with the `LWS` scheduler. The same thing occurs with the B factor, for example, when the allocation of variation function request for the entries with factor B equals -1, it will search measurements with the number of GPUs equals zero, and when it receives one, it will find entries with one worker per GPU.

The allocation of variance developed function, as previously reported, is design to work only with four factors. It receives the raw input data loaded from the CSV file, group all the data with the same combination of Scheduler, number of GPU workers, block size, and inner block size and calculate the mean and the standard deviation. After that, it builds the sing table using the `selected_values` function and calculates the allocation of variance for each factor, returning the data frame as specified from the book.

```

# Function to extract only the derired information from data
selected_values <- function(data, A, B, C, D){
    # Default Options equal -1 on table
    sched <- "lws"
    ngpu <- 0
    size <- 320
    ib <- 32

    # Other Options equal 1 on table
    if(A==1){
      sched <- "dmda"
    }

    if(B==1){
      ngpu <- 1
    }

    if(C==1){
      size <- 640
    }

    if(D==1){
      ib <- 64
    }

      # The data can have other measurements, filter it
    value <- data %>% filter(Scheduler==sched,
                             WGPUS==ngpu,
                        Size==size,
                             IB==ib)

    return(list(value$Mean, value$Sd))

}


alloc_vari_4_fact <- function(data){
    # Values for the signs table
  x <- seq(-1, 1, 2)



  data %>% group_by(Scheduler, WGPUS, Size, IB) %>%
     summarize(N=n(), Mean=mean(Time), Sd=sd(Time)) -> data_mean

  d1 <- expand.grid(A = x, B = x, C = x, D = x)
  d1 %>% rowwise() %>% mutate(AB=A*B,
          AC=A*C,
          AD=A*D,
          BC=B*C,
          BD=B*D,
          CD=C*D,
          ABC=A*B*C,
          ABD=A*B*D,
          ACD=A*C*D,
          BCD=B*C*D,
          ABCD=A*B*C*D,
          y = selected_values(data_mean, A, B, C, D)[[1]],
          sd = selected_values(data_mean, A, B, C, D)[[2]]
          ) %>% ungroup() -> semi_table

   semi_table %>%
          mutate(xA = A*y,
                 xB = B*y,
                 xC = C*y,
                 xD = D*y,
                 xAB = AB*y,
                 xAC = AC*y,
                 xAD = AD*y,
                 xBC = BC*y,
                 xBD = BD*y,
                 xCD = CD*y,
                 xABC = ABC*y,
                 xABD = ABD*y,
                 xACD = ACD*y,
                 xBCD = BCD*y,
                 xABCD = ABCD*y)  %>%
        summarize(TA=sum(xA)/16,
                  TB=sum(xB)/16,
```

```r
                    TC=sum(xC)/16,
                    TD=sum(xD)/16,
                    TAB=sum(xAB)/16,
                    TAC=sum(xAC)/16,
                    TAD=sum(xAD)/16,
                    TBC=sum(xBC)/16,
                    TBD=sum(xBD)/16,
                    TCD=sum(xCD)/16,
                    TABC=sum(xABC)/16,
                    TABD=sum(xABD)/16,
                    TACD=sum(xACD)/16,
                    TBCD=sum(xBCD)/16,
                    TABCD=sum(xABCD)/16

  ) %>% data.frame() -> qs

  qs_vector <- as.numeric(qs[1,])

  sum(qs_vector^2)*16 -> total

  (qs_vector^2*16) / total * 100 -> influ

  semi_table$y <- sprintf("%.2f", semi_table$y)
  semi_table$sd <- sprintf("%.2f", semi_table$sd)

  semi_table %>% data.frame() -> sing

  x <- c(sprintf("%.2f", qs_vector), "", "")
  f <- c(sprintf(" %.2f%% ", influ), "", "")

  result <- rbind(rbind(sing, x), f)
  return(result)
}
```

4.1.4 Apply function on measurements

We apply the function and get the 18th row, that contains the allocation of variation for each factor and their combination.

```r
alloc_var_tf17_draco <- alloc_vari_4_fact(data_tf17_draco)[18,]
alloc_var_tf17_tupi <- alloc_vari_4_fact(data_tf17_tupi)[18,]
alloc_var_tf16_draco <- alloc_vari_4_fact(data_tf16_draco)[18,]
```

4.1.5 Summarize values

Let's create a table with all variation values adding the source of values and rearrange the columns order.

```r
alloc_var_all <- rbind(alloc_var_tf17_draco %>% mutate(Source = "TF17_draco"),
                       alloc_var_tf17_tupi %>% mutate(Source = "TF17_tupi"),
                       alloc_var_tf16_draco %>% mutate(Source = "TF16_draco"))

alloc_var_all %>% select(Source, A:ABCD)
```

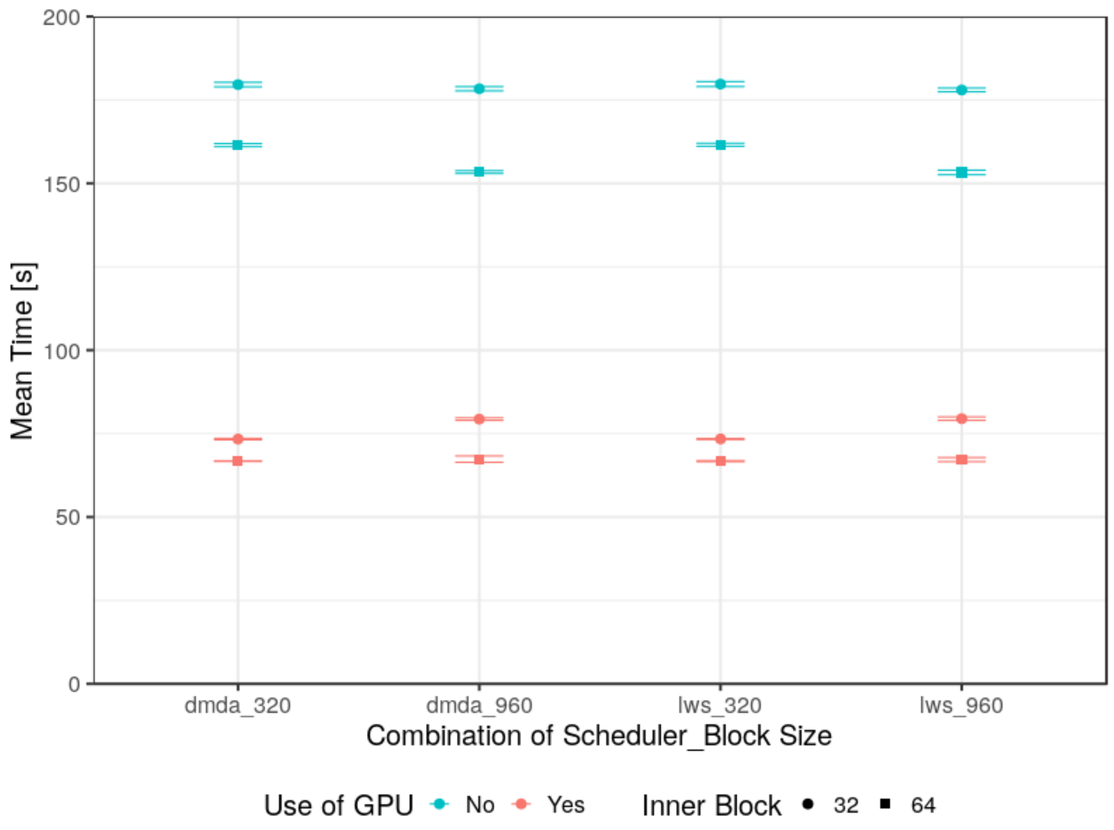
```
      Source       A        B       C       D      AB      AC      AD      BC
1 TF17_draco  0.02%   83.59%   3.62%   9.36%   0.00%   0.00%   0.00%   3.13%
2  TF17_tupi  0.00%   95.26%   0.13%   2.93%   0.00%   0.00%   0.00%   1.34%
3 TF16_draco  0.00%   92.88%   0.27%   2.95%   0.00%   0.00%   0.00%   3.52%
       BD      CD     ABC     ABD     ACD     BCD    ABCD
1   0.15%   0.07%   0.00%   0.00%   0.00%   0.05%   0.00%
2   0.10%   0.14%   0.00%   0.00%   0.00%   0.10%   0.00%
3   0.23%   0.03%   0.00%   0.00%   0.00%   0.10%   0.01%
```

The data indicates that the factor A (scheduling) had almost none impact on all conditions, including all combinations with it. The factor B (use of GPU) had the most performance influence on all the circumstances, reaching 95.26% using the TF17 on the `TUPI` case. The C factor (block size) presented a small impact on TF17 `Draco` case and almost none on the other two, yet its combination with B, the BC case, presented small variations on all situations. The last isolated factor, D, the inner block size, exhibited medium allocation of variation on the first case and a modest on the other two. All the additional combinations presented a minimal allocation of variance.

4.1.6 Extra visualization for step one

We also create this visualization for one data of step one to analyze the values in their real magnitudes and not only percentages of variation. Where in the Y-axis there is the mean execution time for that configuration, with 99% of confidence (using $t_{99}(9) = 3.25$), and in the X-axis there are multiple configurations in the form of a scheduler, block size and inner block size. Also, the colors indicate the number of workers per GPU used, blue for zero and red for one. We can observe that the principal difference is the number of workers per GPU, in this case, zero if the GPU is not used or one if there is only one worker per GPU. Although it seems that the time difference of the block size and inner block size factors are small, all cases present disjoint points considering the confidence interval.

```
data_tf17_tupi %>% filter(Scheduler %in% c("dmda", "lws") ) %>%
        filter(WGPUS<2, Size %in% c(320, 960)) %>%
        group_by(Scheduler, WGPUS, Size, IB) %>%
        summarize(N=n(), Mean=mean(Time), Sd=sd(Time)) %>%
        mutate(T=paste0(Scheduler, "_", Size)) %>%
        ggplot(aes(x=T, y=Mean, color=factor(WGPUS))) +
        geom_errorbar(aes(ymin=Mean-3.25*Sd/sqrt(N), ymax=Mean+3.25*Sd/sqrt(N)),
    width=.2) +
        geom_point(aes(shape=factor(IB)), size=3) +
        scale_shape_manual(values=c(19, 15))+
        scale_color_manual(labels=c("No", "Yes"), values = c("#00BFC4", "#F8766D
  ")) +
        scale_y_continuous(limits = c(0, 200), expand=c(0.0,0), name = "Mean
  Time [s]") +
        theme_bw(base_size=20) +
        xlab("Combination of Scheduler_Block Size") +
        theme(legend.position="bottom") +
        guides(shape=guide_legend(title="Inner Block"),
               color=guide_legend(title="Use of GPU"))
```



We can check the real values for the mean times of the configurations. Its also important to mention that the variance impact for the other factors was obscured by the use of GPU, as it had such great influence on the system performance. However, the variation of these other parameters can also be considered significant. The `TF17 Draco` case is a great example where we can see the difference with the other factors, the C factor had 3.62%, and D had a 9.36% allocation of variation.

4.2 Second Step

After the first step, we conclude that the scheduler parameter had almost no impact, so we decided to concentrate the next experiment on the other factors. Also, after we checked that the use of GPU was the most impacting factor, we executed all the subsequent tests with at least one worker per GPU, removing the level of zero workers (not using GPUs). The parameters for the next batch of experiments are the following. (i) Workers per GPU, from one to four; (ii) Block size, with levels 360, 640 and 960; (iii) Inner block size, with values 32 and 64. Moreover, to reduce the number of experiments, and because we didnt notice any drastic difference between machines, we executed all the next experiments on the `TUPI` machine. Each combination is performed 30 times with the sequence between configurations randomized; the generated tests' order is available at the public data. The files for each experiment are separated per matrix on files: `s2_tf16.txt`, `s2_tf17.txt`, `s2_mk12.txt` and `s2_ch7.txt`.

4.2.1 Read New Data

Let's read all new time files with the previously defined function.

```
data_tf16 <- read_measurements("data/s2_tf16.txt")
data_tf17 <- read_measurements("data/s2_tf17.txt")
data_mk12 <- read_measurements("data/s2_mk12.txt")
data_ch7 <- read_measurements("data/s2_ch7.txt")
```

4.2.2 Calculate Time Mean

Let's calculate the mean time and standard deviation for each group of scheduler, workers per GPU, block size and IB. Also, let's create the `mat` column with the matrix used.

```
m_tf16 <- data_tf16 %>% group_by(Scheduler, WGPUS, Size, IB) %>%
          summarize(N=n(), Mean=mean(Time), Sd=sd(Time)) %>%
          arrange(Mean) %>% mutate(mat="tf16")

m_tf17 <- data_tf17 %>% group_by(Scheduler, WGPUS, Size, IB) %>%
          summarize(N=n(), Mean=mean(Time), Sd=sd(Time)) %>%
          arrange(Mean) %>% mutate(mat="tf17")

m_mk12 <- data_mk12 %>% group_by(Scheduler, WGPUS, Size, IB) %>%
          summarize(N=n(), Mean=mean(Time), Sd=sd(Time)) %>%
          arrange(Mean) %>% mutate(mat="mk12")

m_ch7 <- data_ch7 %>% group_by(Scheduler, WGPUS, Size, IB) %>%
          summarize(N=n(), Mean=mean(Time), Sd=sd(Time)) %>% filter(WGPUS>0) %>%
          arrange(Mean) %>% mutate(mat="ch7")
```

4.2.3 Define a easy plot function

Let's create an easy to call plot function to visualize these data. In the X-axis there is the number of workers per GPU, and in the Y-axis the factorization time means, with a confidence of 99% (using $t_{99}(29) = 2.756$). Also, the circle points represent the use of 32 as the inner block size and the squares the value 64. The line shape indicates the value for the block size; where the continuous line is 320, the dashed is 640, and the dotted is 960. Also, the colors reinforce the combination of block size and the inner block size (colors makes easier to check the block size and the inner block size combination). The parameters for the function are:

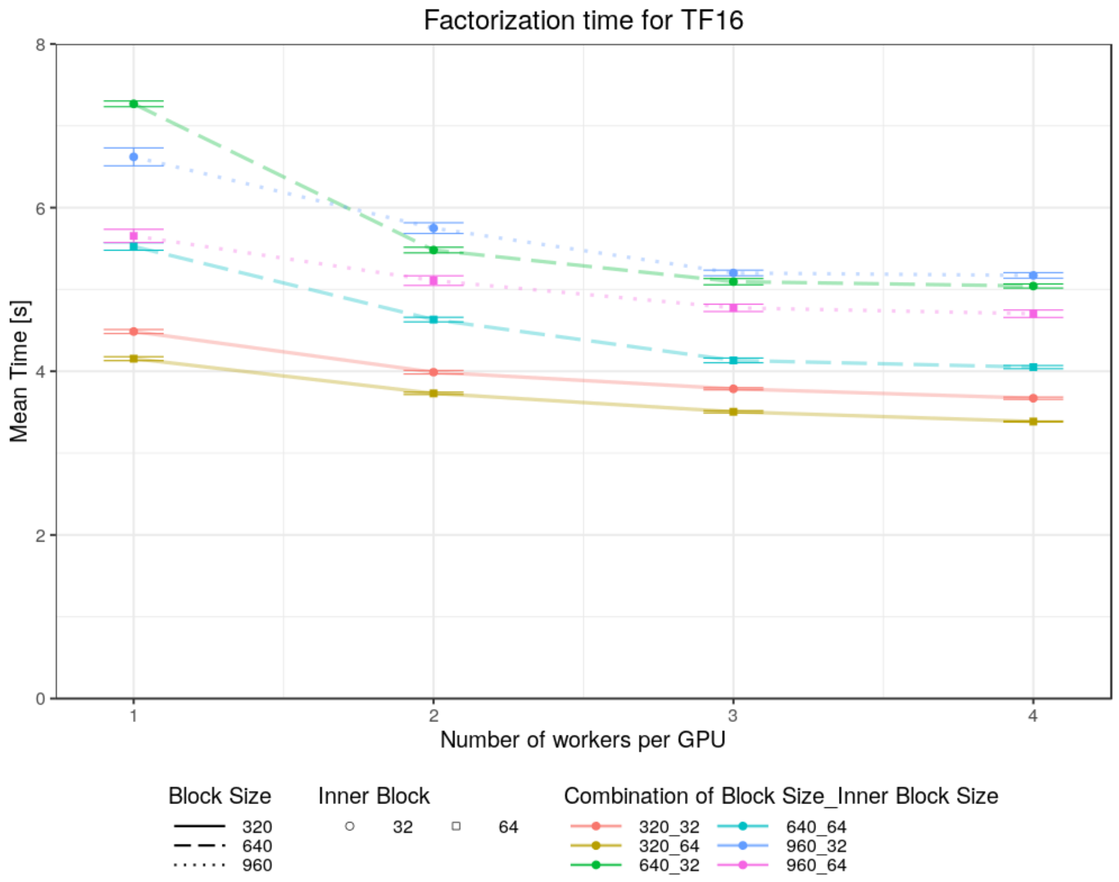
- data: The data used in the output format of `read_measurements` function.
- up: The upper value for the Y-axis (mean time)
- title: The name of the matrix used to add in the title.

```
step2_plot <- function(data, up=NA, title=""){

data %>% filter(WGPUS>0, Size<1000) -> final_data

final_data %>% mutate(T=paste0(Size, "_", IB)) %>%
      ggplot(aes(x=WGPUS, y=Mean, group=T, color = T)) +
      geom_line(aes(linetype=factor(Size)), size=1.5, alpha=0.35) +
      geom_errorbar(aes(ymin=Mean-2.756*Sd/sqrt(N),
                        ymax=Mean+2.756*Sd/sqrt(N)),
                    width=.2) +
      geom_point(aes(shape=factor(IB), fill = T), size=3) +
      scale_y_continuous(limits = c(0, up), expand=c(0.0,0), name = "Mean Time [s
    ]") +
      scale_x_continuous(name = "Number of workers per GPU") +
      scale_linetype_manual(values=c("solid", "longdash", "dotted"))+
      scale_shape_manual(values=c(21, 22))+
      ggtitle(paste0("Factorization time for ", title)) +
      theme_bw(base_size=20) +
      theme(legend.position="bottom",
            plot.title = element_text(hjust = 0.5),
            legend.key.width = unit(2,"cm")) +
      guides(color=guide_legend(title="Combination of Block Size_Inner Block Size
    ",
                                title.position="top", order = 3, ncol = 2),
             fill=guide_legend(title="Combination of Block Size_Inner Block Size"
    ,
                               title.position="top", order = 3, ncol = 2),
             shape=guide_legend(title="Inner Block", title.position="top",
                                order = 2),
                linetype=guide_legend(title="Block Size",
                                      title.position="top", order = 1,
                                      ncol = 1,
                                      override.aes = list(alpha = 1, size=1))
            )
}
```

4.2.4 Generate the Time plots

Let's call the plot function in all data and analyze. First, let's check TF16.

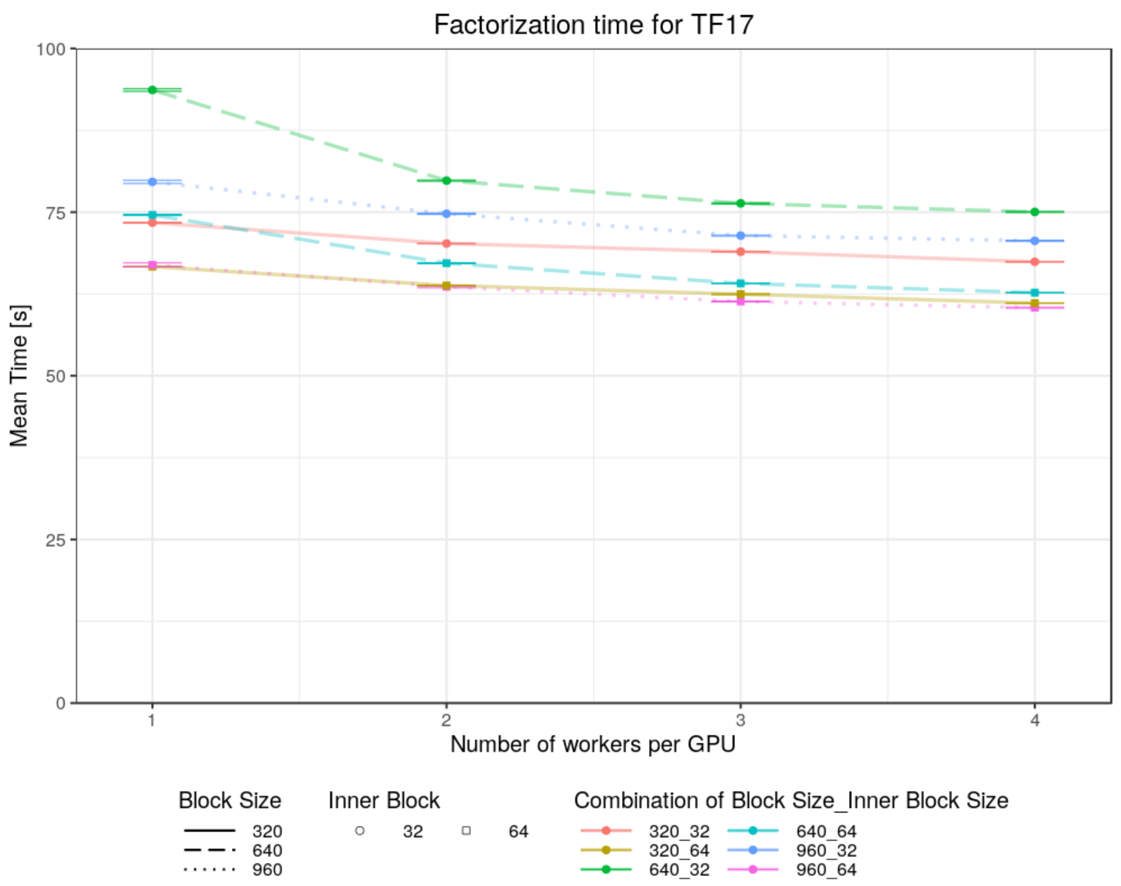
```
step2_plot(m_tf16, 8, "TF16")
```



In the last figure, we can check the mean factorization time results for the execution using the TF16 matrix. The combination 320 64 is the best one independent (with statistic significance) of the number of GPUs. Also, there is some performance increase from one worker per GPU to two workers, and then three; however, when increasing to four workers the minimal difference is inside the confidence interval.

Let's check the TF17 case:

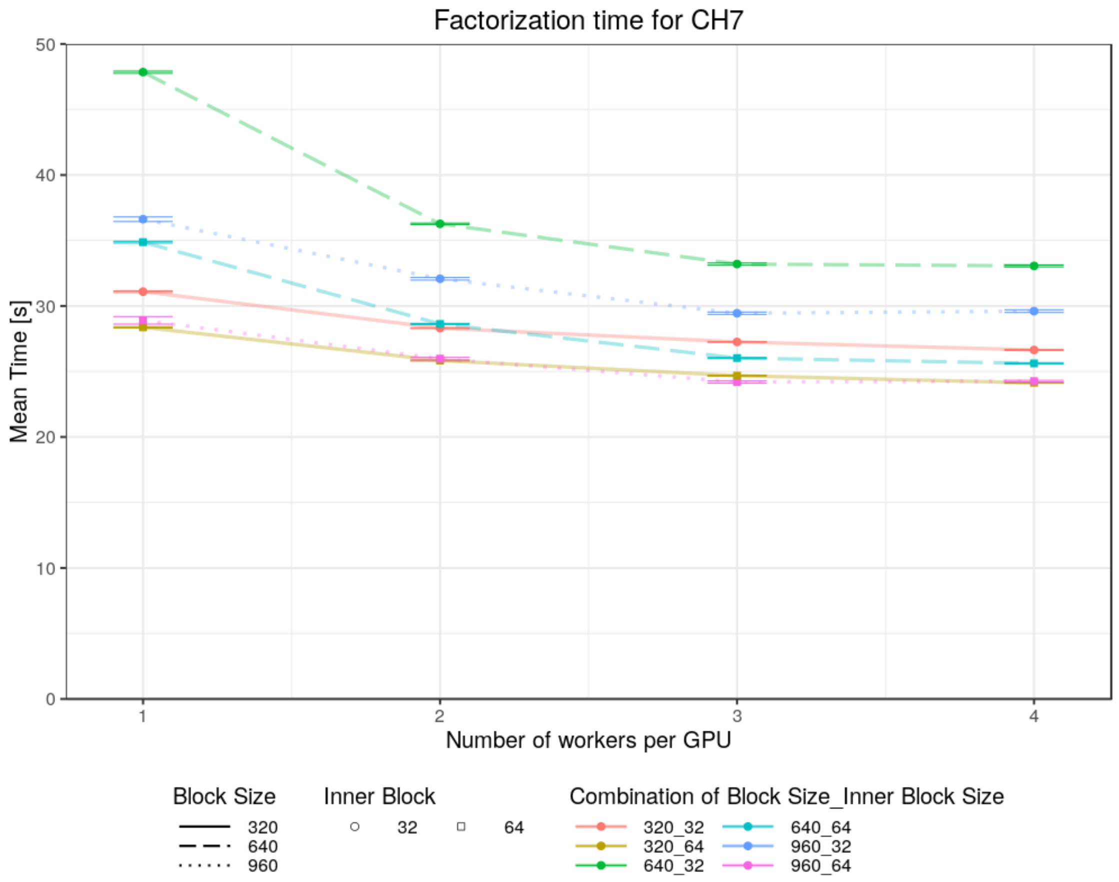
```
step2_plot(m_tf17, 100, "TF17")
```



Similar to the TF16 case, the combination 320 64 was the best one independent of the number of GPU workers. However, in this case, it shared the position with the combination 960 64 that had a statistically tied using the confidence interval. Moreover, the worst combinations, in this case, is using the inner block size of 32. The performance difference when increasing the number of workers per GPU is more apparent from one to two workers, while from 3 to 4 none significant statistical difference in performance is noted.

Let's check the CH7 case:

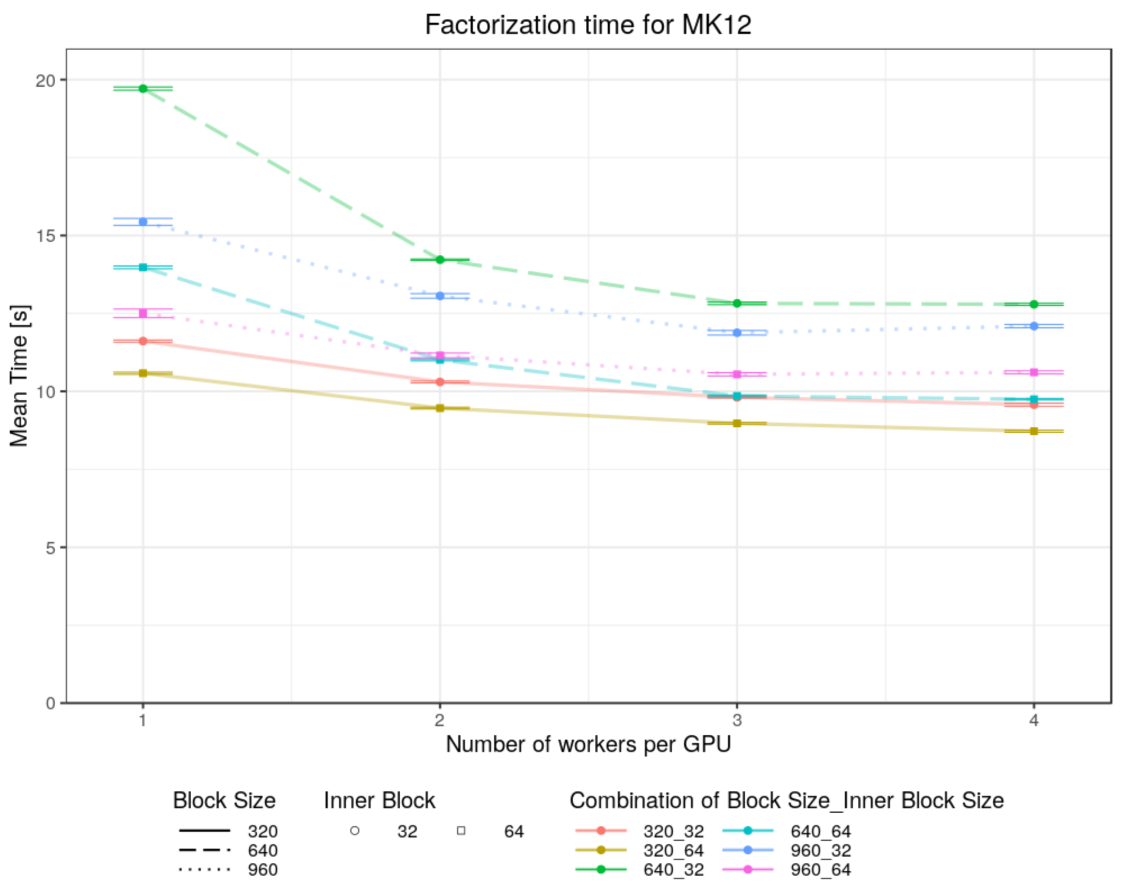
```
step2_plot(m_ch7, 50, "CH7")
```



This case has similar results to the TF17 one. It worth to mention that these matrices were the larges ones and also had the biggest execution time among the tested ones. The combinations 320 64 and 960 64 were the best ones, in a statistical tie, independent of the number of workers per GPU. The worst cases were the three combinations using the inner block size of 32, and there is a significant performance increase from one to two workers per GPU.

And lets check the last matrix:

```
step2_plot(m_mk12, 21, "MK12")
```



It had similar results with the TF16 matrix, both ones were the smallest matrices and had the lower execution times from the tested ones. The case 320 64 was the best one alone while the 640 32 was the worst one in all cases independent of the number of workers per GPU.

5 Conclusion and Next Steps

In this investigation, we studied a set of execution parameters of the software package `QR_Mumps` that run over the StarPU runtime. We selected four parameters, two from each software to understand how much they change the final application performance. In our experiments, we verified that some of the parameters had much more impact than others, and report the best values for each one.

From the four select parameters, StarPU scheduler, StarPU number of workers per GPU, `QR_Mumps` block size, and `QR_Mumps` inner block size, we verified the individual impact using the allocation of variation method. We concluded that the StarPU scheduler did not influence the performance, while the StarPU number of workers per GPU had the most impact. The block size and the inner block size had some effect, where different select values presented statistical difference. For this reason, we not considered the StarPU scheduler in the next experimental step.

After the first step of experiments, we evaluate all the options for the selected three factors; Number of Workers per GPU, Block Size, and inner block size. The behavior of the parameter number of workers per GPU indicates that the GPUs got saturated after two workers, however, adding more workers didnt impact negatively in the performance, any value greater than two is sufficient here. Considering the block size, the only option that was the best regarding any matrix was 320. In all cases, the inner block size of 64 was better than the equivalent 32 variant. In conclusion, the best combination of the proposed parameters is any scheduler, four workers per GPU, block size of 320, and inner block size of 64.

In future work, we intend to investigate why these behaviors occur. This goal could be achieved with the use of executions traces and performance analysis toolkits like StarVZ. Also the investigation of other parameters of both StarPU and `QR_Mumps` is also possible to continue to improve the performance of the application without changing the hardware or the source code.